A Fast Parallel Solver for the Forward Problem in Electrical Impedance Tomography

Markus Jehl*, Andreas Dedner, Timo Betcke, Kirill Aristovich, Robert Klöfkorn, and David Holder

***Abstract*—Electrical impedance tomography (EIT) is a noninvasive imaging modality, where imperceptible currents are applied to the skin and the resulting surface voltages are measured. It has the potential to distinguish between ischaemic and haemorrhagic stroke with a portable and inexpensive device. The image reconstruction relies on an accurate forward model of the experimental setup. Because of the relatively small signal in stroke EIT, the finite-element modeling requires meshes of more than 10 million elements. To study the requirements in the forward modeling in EIT and also to reduce the time for experimental image acquisition, it is necessary to reduce the run time of the forward computation. We show the implementation of a parallel forward solver for EIT using the DUNE-FEM C++ library and demonstrate its performance on many CPU's of a computer cluster. For a typical EIT application a direct solver was significantly slower and not an alternative to iterative solvers with multigrid preconditioning. With this new solver, we can compute the forward solutions and the Jacobian matrix of a typical EIT application with 30 electrodes on a 15-million element mesh in less than 15 min. This makes it a valuable tool for simulation studies and EIT applications with high precision requirements. It is freely available for download.**

***Index Terms*—Electrical impedance tomography (EIT), finite-element solver, forward problem, parallel computing.**

I. Introduction

A. Applications of Electrical Impedance Tomography

ELECTRICAL impedance tomography (EIT) is an imaging modality in which low-frequency currents are applied to the surface of the body under examination and the resulting surface potentials are measured. Doing this for a defined protocol of applied current patterns gives a current-to-voltage or Neumann-to-Dirichlet (NtD) map, which is used in the inverse problem for creating an image. Many medical applications are envisaged for EIT, some of which have already been successfully applied. Monitoring lung ventilation is probably the most mature field of EIT, being pioneered by Barber and Brown [1] in the early 1980s (for a review of lung EIT research, see [2]). Another well-validated application of EIT is the analysis of gastric emptying, a field which is reviewed for instance in [3].

EIT has the potential to be used in neuroscience to image fast neural activity by measuring the impedance change due to the opening of neuron's ion channels [4]. It would be the first time that neuronal activity can be directly recorded noninvasively. Furthermore, EIT can potentially be used for fast and inexpensive stroke type differentiation and improve the outcome for patients. Feasibility studies of stroke EIT include [5], where the signal change during global ischaemia was measured with scalp electrodes, and [6] and [7], where the effect of modeling errors on the image quality is discussed.

B. Numerical Solvers for the Complete Electrode Model

Most research groups in EIT currently use the Electrical Impedance Tomography and Diffuse Optical Tomography Reconstruction Software EIDORS [8], which is programmed in MATLAB. EIDORS provides a set of useful features, such as 2-D and 3-D forward simulations and an extensive set of reconstruction algorithms, visualization functions, and more. Horesh *et al.* [9] adapted EIDORS with different preconditioners and more efficient routines, resulting in a version called SuperSolver, which is still used in our group at UCL. For large meshes, however, MATLAB suffers from a lack of efficient parallel programming possibilities, which makes the computation of forward solutions a lengthy task.

Borsic *et al.* [10] moved the forward and the Jacobian calculations (not the assembly of the system matrix, though) to sparse parallel direct solver library PARDISO [11] to surpass these limitations. They were able to improve the speed for forward simulations about 5.3 fold compared to Horesh *et al.* [9]. They used it on meshes with around half a million elements. On larger meshes, direct solvers require large amounts of memory that normally limits the mesh size that can be computed. Furthermore, we show in this paper, that the assembly of the direct solver is much slower than that of a good preconditioner, resulting in faster execution times for iterative methods depending on the number of unique current injection patterns. In particular, the algebraic multigrid preconditioner has been shown to improve the solution time significantly [12]. Graphics processing unit (GPU) based computations have already successfully been applied to the calculation of the Jacobian matrix [13], where fast access to the memory is paramount. A different approach to the forward modeling in EIT was done by using boundary elements [14], a technique that requires the head to be modeled as enclosed surfaces of the different tissues with fixed conductivity.

Manuscript received January 22, 2014; accepted July 19, 2014. Date of publication July 23, 2014; date of current version December 18, 2014. *Asterisk indicates corresponding author.*

*M. Jehl is with the Department of Mathematics and the Department of Medical Physics and Bioengineering, University College London, London WC1E 6BT, U.K. (e-mail: markus.jehl.11@ucl.ac.uk).

A. Dedner is with the Centre for Scientific Computing/Mathematics Institute, University of Warwick, Coventry CV4 7AL, U.K. (e-mail: A.S.Dedner@warwick.ac.uk).

T. Betcke is with the Department of Mathematics, University College London, London WC1E 6BT, U.K. (e-mail: t.betcke@ucl.ac.uk).

K. Aristovich and D. Holder are with the Department of Medical Physics and Bioengineering, University College London, London WC1E 6BT, U.K. (e-mail: k.aristovich@ucl.ac.uk; d.holder@ucl.ac.uk).

R. Klöfkorn is with the Computational and Information Systems Laboratory, University Corporation for Atmospheric Research, Boulder, CO 80305 USA (e-mail: robertk@ucar.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TBME.2014.2342280

This works well for piecewise homogeneous media, but not for complicated heterogeneous geometries like the human head.

C. Main Results

We have implemented a parallel forward solver with the complete electrode model (CEM) in C++ using message passing interface (MPI). Comparing the performance of a direct solver and an iterative solver with multigrid preconditioning, we found that for a typical EIT application the direct solver is significantly slower and no alternative to multigrid implementations. Using this parallel solver, the forward solutions and Jacobian matrix are computed on one processor more than twice as fast than using EIDORS. Furthermore, on ten processors, the total runtime reduces more than ten fold. This makes the presented solver an invaluable tool for forward simulations on large meshes. It is freely available for download (instructions on http://users.dune-project.org/projects/dune-peits/wiki).

We highlight two applications of the presented solver that have been published separately in more detail and illustrate the impact the developed software makes on the time efficiency of both, simulation studies and experimental imaging in EIT.

II. Mathematical Formulation of the CEM

The commonly used CEM accounts for two observed effects, electrodes shunting current due to their high conductivity and a voltage drop at the interface of the electrodes and the skin, which is due to an electrochemical effect. The applied current is constant on each electrode $(\Gamma_l)_{l=1,\dots,M}$ and vanishes on the surface between electrodes $\Gamma \setminus \bigcup_{l=1}^{M} \Gamma_l$. It can thus be represented as a vector $(I_l)_{l=1,\dots,M}$ in $\mathbb{R}^M$ satisfying the condition $\sum_{l=1}^{M} I_l = 0$. According to the CEM, the voltage potential u then solves

$$\nabla \cdot (\sigma \nabla u) = 0 \quad \text{in } \Omega \tag{1}$$

$$\int_{\Gamma_l} \sigma \frac{\partial u}{\partial \nu}\, d\Gamma_l = I_l \quad l = 1, \dots, M \tag{2}$$

$$u + z_l \sigma \frac{\partial u}{\partial \nu} = U_l \quad \text{on } \Gamma_l, \quad l = 1, \dots, M \tag{3}$$

$$\sigma \frac{\partial u}{\partial \nu} = 0 \quad \text{on } \Gamma \setminus \bigcup_{l=1}^{M} \Gamma_l \tag{4}$$

where $(z_l)_{l=1,\dots,M} \in \mathbb{R}^M$ is the positive contact impedance and $(U_l)_{l=1,\dots,M} \in \mathbb{R}^M$ is the vector of the voltage potentials on the electrodes abiding to the grounding condition $\sum_{l=1}^{M} U_l = 0$. ν is the outward unit normal to Γ and $\sigma \in L^\infty(\Omega)$ is the positive real conductivity. The CEM is proven to have a unique solution $u \in H^1(\Omega)$, which depends continuously on $I \in L^2(\Gamma)$ satisfying $\int_\Gamma I ds = 0$ [15].

The weak formulation of the CEM is obtained by integrating (1) over Ω with a set of test functions v in H^1, applying Green's formula and substituting the electrode potentials U_l using (3)

$$\int_\Omega \sigma \nabla v \nabla u + \sum_{l=1}^{M} \frac{1}{z_l} \int_{\Gamma_l} vu - \sum_{l=1}^{M} \frac{1}{z_l |\Gamma_l|} \int_{\Gamma_l} v \int_{\Gamma_l} u = \sum_{l=1}^{M} \frac{1}{|\Gamma_l|} \int_{\Gamma_l} v I_l. \tag{5}$$

One thing to notice in this weak formulation is the uncommon third term of the type $\int_{\Gamma_l} u \int_{\Gamma_l} v$. To facilitate the parallel assembly of this term and to reduce communication between processes, we have to ensure that each electrode is not split onto different partitions (see Section III-D for the implementation).

Proposition II.1: The system (5) is uniquely solvable if a ground condition is applied.

Proof: To prove that this system is positive, and thus, uniquely solvable, we have to show that the second term of the left-hand side (LHS) is larger than the third term. Replacing u by v and multiplying with z_l, we get from Cauchy–Schwarz

$$\int_{\Gamma_l} v^2 \geq \frac{1}{|\Gamma_l|} \left(\int_{\Gamma_l} v \right)^2 \tag{6}$$

with $v = \text{const.}$ leading to equality. Because a constant v sets the first term of the LHS to zero as well, the system is only positive semidefinite. Thus, we need an additional constraint to make the system uniquely solvable. Setting a ground condition achieves this. ■

Different grounding conditions can be applied. We decided to set one surface node to 0 V by applying a Dirichlet boundary condition.

Once the forward solutions are computed, most EIT inversion algorithms require the so-called *Jacobian matrix* that translates a change in conductivity to a change in measured voltages by linearization at the simulated conductivity distribution. Our approach to the calculation of the Jacobian matrix is the lead (or adjoint) fields method (which is derived, e.g., in the Appendix of [16])

$$\delta V_{dm} = -\int_\Omega \delta\sigma \nabla u(I^d) \cdot \nabla u(I^m)\, dV \tag{7}$$

where $u(I^d) \in H^1(\Omega)$ is the electric potential emerging when the drive current I^d is applied to the electrodes and $u(I^m) \in H^1(\Omega)$ the electric potential when a unit current is applied to the two measurement electrodes. $\delta V_{dm} \in \mathbb{R}$ is then the linearly approximated voltage change between the two measurement electrodes when the conductivity changes by $\delta\sigma \in L^\infty(\Omega)$.

III. Large-Scale EIT Solver Based on Dune

Currently, most medical application of EIT monitor large physiological changes in the trunk such as lung ventilation and gastric emptying. Such tasks are not very challenging on the modeling, since the geometry is comparatively simple and the regions of the conductivity change are large, resulting in a high signal-to-noise ratio. Our aim is the detection of stroke, where we want to image a comparably small perturbation that is "hidden" under a highly resistive layer of skull and a highly conductive layer of cerebrospinal fluid. This imposes much higher

precision requirements on the forward modeling than is commonly necessary in EIT.

Preliminary results from an ongoing convergence analysis for finite-element meshes based on the same segmentation of a head, revealed that the optimal mesh size exceeds 10 million elements and might be in the region of 15 to 20 million elements. For the rat brain, which is geometrically less complex than the human head with all its different tissues, the optimal mesh size is 7-million elements [17]. Using a 15-million element mesh in EIDORS, it takes 990 s to set up the system matrix and another 936 s to solve for each unique current pattern using an incomplete LU preconditioned conjugate gradients algorithm.

Our aim was to build a solver that can run on parallel machines and clusters to reduce the computational time of the image acquisition process in EIT. We sometimes use the abbreviation *PEITS* (*P*arallel *EIT S*olver) in this paper and in the source code. A guide on how to install and use the solver can be found on the project wiki (http://users.dune-project.org/projects/dune-peits/wiki).

A. Overview of Dune

The solver we present here is based on DUNE. The Distributed and Unified Numerics Environment, DUNE, is a grid-based C++ toolbox for solving partial differential equations. DUNE includes the discretization module DUNE-FEM, which allows implementations of finite-element solvers for parallel computers. It provides functions to implement local grid adaptivity, dynamic load balancing, and higher order discretization schemes [18]. Apart from native implementations of conjugate gradients solvers it also provides interfaces to the solvers and preconditioners of the DUNE-ISTL module, UFMPACK [19] for unsymmetrical problems and PETSc [20], which has an extensive collection of solvers and preconditioners. DUNE-FEM supports two types of parallelism, the MPI and pthread. DUNE is licensed under the GNU General Public Licence version 2.0, and thus, free to use for everyone.

We decided to use DUNE-FEM because it is a slim, template-based, and thus, versatile C++ library that allows us to implement the CEM, which has an uncommon weak formulation and is thus not easily implementable in most finite-element libraries. Furthermore, DUNE-FEM provides an interface to all preconditioners and solvers we require and supports tetrahedral elements. The module is still in development and was flexibly adjusted to our needs.

B. Implementation of the CEM in Dune-Fem

The code is structured in different files that contain classes, structs and functions for specific tasks. The main file is `dune_peits.cc`, which performs the following important steps in this order.

1) Loading the mesh and partitioning it. If the mesh was already partitioned before, those partitions are loaded by the parallel processes directly.
2) The electrode positions are loaded into a struct. This struct has query functions that evaluate if a given element belongs to an electrode, return contact impedances of specific electrodes and more.
3) The current protocol is read from the specified file. Upon reading the protocol it is disassembled into unique injections. The solution for each unique current injection is computed just once. This reduces the number of required forward solutions for a standard EIT protocol with around 1000 lines to around 60.
4) The system matrix is assembled. The function that computes the system matrix entries is located in the file `elliptic.hh`.
5) In a for-loop, the following steps are performed for each unique current injection:
   a) The right-hand side of the weak formulation is assembled using a function in `rhs.hh`.
   b) The CG solver computes the resulting electric potential and the result is stored in a vector.
   c) If specified in the parameter file, the first solution is written to a VTK file for visual inspection.
6) In a second for-loop, the following steps are performed for each line in the current protocol.
   a) Trace back which solutions correspond to the drive current and measurement current of this protocol line.
   b) If selected in the parameter file, compute the measured voltage and save to a binary file.
   c) Compute the row of the Jacobian matrix using the forward solutions for the drive and measurement current and save it to a binary file.

C. Methods

Unless otherwise noted we computed all run times on a head mesh with different conductivities for the scalp, skull, cerebrospinal fluid (CSF), white matter, gray matter, and dura mater (see Fig. 1). The meshes were created from a CT and MRI scan of the same patient's head using the meshing toolbox of CGAL [21]. In particular, the assembly of the system matrix will be slower, the more elements are part of an electrode. To make sure that the results we are presenting here can be compared to each other, we fixed the ratio of electrode elements to other elements by having a constant element size throughout the mesh. For real applications it is much better to refine the mesh around the electrodes and use much larger elements toward the center of the head.

To measure the parallel scalability of the code, we commonly plot the efficiency, which is calculated as follows for p parallel processes

$$\text{efficiency}(p) = \frac{\text{runtime}(1)}{\text{runtime}(p) \cdot p}. \tag{8}$$

This efficiency is a value for the strong scaling. For the weak scaling, we want to show how much more elements we can compute in the same time by using more processors. We can thus use the following definition for the efficiency of the weak scaling, where x is a fixed number of elements and runtime(px)

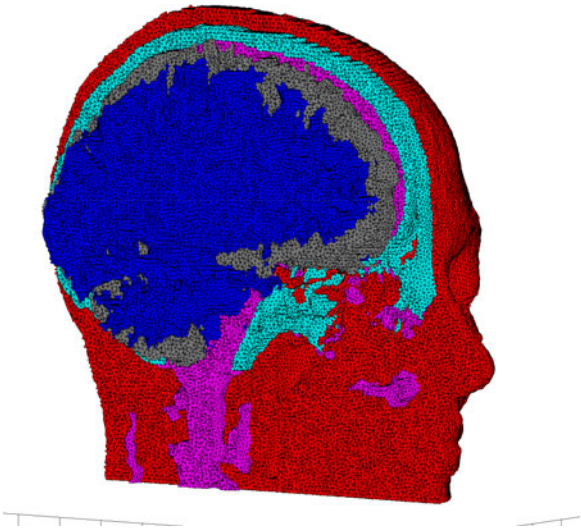

Fig. 1. Layered cut through a 5-m element head mesh with scalp, skull, CSF, and gray and white matter—The mesh was created with CGAL from a segmentation of a CT and an MRI scan of the same person. It also includes parts of the dura mater and air cavities, which are not visible in this image.

means the time it takes to compute px elements on p processors

$$\text{efficiency}(p) = \frac{\text{runtime}(1x)}{\text{runtime}(px)}. \tag{9}$$

All run times were taken on a cluster with five nodes. Each node had two 6 core 2.40-GHz Intel Xeon processors with 12-MB cache and a total of 192 GB of memory. The nodes were connected by a dedicated 1-GB Ethernet switch. PETSc version 3.4.2 and Zoltan version 3.6 were used.

D. Parallel Substructuring

When using a finite-element mesh for the first time, it needs to be partitioned evenly onto the imposed number of parallel processes. DUNE-FEM has a partitioning tool available, but this tool does not enable the user to guide the load-balancing by fixing regions to a specific process. We require the electrodes to be on only one process each, since this facilitates the correct system matrix assembly and minimizes communication between processes. A well documented and established library supporting user-defined load balancing is Zoltan [22]. Zoltan is a parallel C, C++, and Fortran 90 library with a simple object-based interface that is easily adapted to many applications. The Zoltan tool we employed is the hypergraph partitioning. A hypergraph interpretation of a finite-element mesh has the following appearance: each element is a vertex of the hypergraph and the element together with all its neighbors forms one hyperedge. Thus, for a mesh with N elements, the hypergraph has N vertices and N hyperedges, which correspond to the communication requirements of the parallel program. The parallel hypergraph partitioning (PHG) tool in Zoltan allows the user to assign different weights to hyperedges and to fix selected vertices to one section.

When a mesh is loaded in our solver for the first time it will initially be partitioned by the load-balancing function of DUNE-FEM. The resulting parts are then made accessible to Zoltan by translating them into the hypergraph format Zoltan requires. Defining the electrode areas with the query functions `ZOLTAN_NUM_FIXED_OBJ_FN` and `ZOLTAN_FIXED_OBJ_LIST_FN`, Zoltan's PHG partitioner is applied to the mesh in order to optimize the load balancing while ensuring that each electrode is assigned to one process only. Zoltan will return a list of elements that need to be moved from one part to another part. This list is subsequently applied using the load-balancing function of DUNE-FEM, the result of which is illustrated in Fig. 2. These three steps are time consuming and do not scale well in parallel, since most of the time is required for the loading of the mesh and the initial load balance, which is serial (see Table I). Furthermore, the number of elements that need to be transferred between processes varies, but tends to increase as the number of processes increases. Since the partitioning has to be done only once for each finite-element mesh and number of parallel processes, the performance of this operation is not critical. The resulting mesh parts are written into separate DUNE grid files (DGF) that can then be loaded in parallel for each subsequent forward computation on the same mesh. The loading of these partitions takes less than a minute for a 15-million element mesh and has a very good parallel efficiency (see Table II).

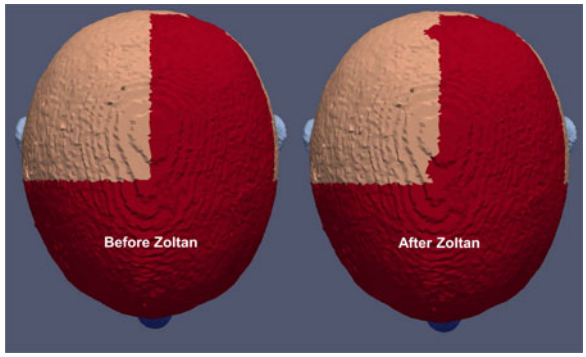

Fig. 2. Partitions before and after Zoltan load balancing—This is a relatively small head mesh with two million elements that is partitioned into four sections. Some electrodes were split onto different processors by the load balancing of DUNE-FEM. The Zoltan partitioner minimizes the number of elements that have to be moved from one process to another, while optimizing the partitions and fixing the electrode areas to one process.

TABLE I
TOTAL TIME REQUIRED FOR THE PARTITIONING OF DIFFERENT MESH SIZES (ALL TIMES ARE IN SECONDS)

Mesh element no	2 m	15 m
12 processes	214	2137
24 processes	250	2405
48 processes	241	2400

E. Assembly of the System

The assembly of the system matrix is done in two mesh iterations. The first iteration stores all elements that belong to an electrode in a 2-D vector `electrodeElements` of

TABLE II
TOTAL TIME REQUIRED TO LOAD PARTITIONS OF DIFFERENT MESH SIZES (ALL TIMES ARE IN SECONDS)

Mesh element no	2 m	15 m
12 processes	5.7	52
24 processes	2.89	26.8
48 processes	1.45	12.8

length $M \in \mathbb{N}$ equal to the number of electrodes with `electrodeElements[k]` containing all elements that constitute electrode k. Furthermore, the overall electrode areas are computed and stored in a vector. Storing the electrode elements is essential to reduce the time for the matrix assembly and later on for the computation of the electrode potentials.

During the second iteration the Laplacian term $\int_{E_i} \nabla v \nabla v \, dE_i$ is added to the system matrix for each element E_i for $i = 1, \ldots, N$ where $N \in \mathbb{N}$ is the number of elements in the mesh. If element E_i constitutes part of an electrode Γ_k a nested loop iterates over all elements of that electrode `electrodeElements[k]` writing each elements contribution to the third term of weak formulation (5) $\frac{1}{z_k |\Gamma_k|} \int_{\Gamma_k} v \, d\Gamma_k \int_{\Gamma_k} v \, d\Gamma_k$ into the system matrix. Additionally, for each element under an electrode the second term of the weak formulation $\frac{1}{z_k} \int_{\Gamma_k} v^2 \, d\Gamma_k$ is assembled.

In a last step, one surface node of the mesh is assigned Dirichlet boundary conditions to make the problem unique. The coordinates of the Dirichlet nodes have to be given to the solver in a mesh specific text file. The solver then evaluates for each node if the coordinates match the given ones and applies the Dirichlet boundary conditions if they do.

Since we are using the numerical solvers available in the PETSc library [20], as discussed in Section III-G, we directly want to assemble the system matrix in the native PETSc sparse row matrix format `MATMPIAIJ`. The difficulty using this format lies in the correct memory preallocation. If PETSc has to reallocate more memory during the matrix assembly the performance can decrease by more than a factor of 50. Due to the DUNE-FEM implementation of the boundaries between parallel partitions, only one side ever knows that there are neighboring elements. Thus, a perfect preallocation can only be achieved by a process in charge of all its interfaces. All other processes underestimate the number of entries in the rows corresponding to communication with elements belonging to other processes. Thus, it is not easily possible to perfectly preallocate the memory for the PETSc sparse row matrix structure in our application. Instead, a first estimate of the maximum number nonzeros per row is preallocated for each mesh and the solver is subsequently run with the option `-info`, which outputs the precise maximal number of nonzeros per row per process and the required number of `mallocs` required. Based on this output, the preconditioning can be further optimized such that no additional `malloc` is required during subsequent solves.

For our realistic head meshes with sizes of up to 15-million elements, we allocated 100 diagonal and 40 off-diagonal entries per row using the PETSc preallocation function `MatMPIAIJSetPreallocation(mat,100, PETSC_NULL,40,PETSC_NULL)` in the corresponding file in the DUNE-FEM library `\dune\fem\misc\petsc\petsccommon.hh`. This approach allocates far more entries than are actually used, but the performance of the matrix assembly is not significantly decreased.

The parallel efficiency of the matrix assembly drops down to around 0.4 for small meshes and 0.5 for large meshes [see Fig. 3(a)]. Even though the efficiency drops, the absolute times to assemble the system matrix still improve on 60 parallel processes (see Table III). The weak scaling shown in Fig. 3(b) indicates that a load of around 0.5-million elements per processor is optimal.

F. Preconditioning

Multigrid methods are known to be very efficient solvers for elliptic boundary value problems, such as the Laplace problem solved in EIT. The underlying principle of multigrid methods is to use several layers of coarseness to guide information exchange rather than having elements exchange information only locally to their direct neighbors. Two general approaches to multigrid methods are geometric multigrid (GMG) and algebraic multigrid (AMG). GMG relies on coarser finite-element meshes with the same geometry, which are not easily created in EIT because of the complicated geometry that cannot simply be coarsened. AMG on the other hand does not require any geometric information and constructs the coarser levels directly from the system matrix, which makes it very adaptable to different problems. With a reduced tolerance, multigrid methods can efficiently be used as preconditioners for interative solvers.

Since the problem (5) solved in EIT is a Laplace problem with a compact perturbation, AMG is the most efficient preconditioner we know of. Through PETSc, we have an interface to two very good AMG implementations, BoomerAMG from Hypre [23] and ML from Trilinos [24]. We compared the performance of the two AMG implementations on two different mesh sizes using the default settings of the respective preconditioner. In Table IV, we list the performance of the two AMG implementations with the default settings on two different sized meshes. The assembly of the ML preconditioner is always faster than that of BoomerAMG. ML preconditioning also results in a faster convergence below 10^{-12} relative residual for the CG solver in most of the cases.

The efficiency of the setup of the AMG preconditioner of Trilinos was tested on two different sized finite-element meshes. As expected, we see a better parallel performance on the large mesh, due to the larger ratio of computation over communication [see Fig. 4(a)]. The weak scaling [see Fig. 4(b)] indicates that a load of approximately $0.5 - 1$ million elements per processor is optimal. Furthermore, it is interesting to compare the weak scaling results of the ML assembly with the weak scaling results for the CG solver [see Fig. 5(b)] and the iteration count of the solver (see Table V), which is a measure of the mesh complexity. The different mesh complexities explain the nonsmooth weak scaling results.

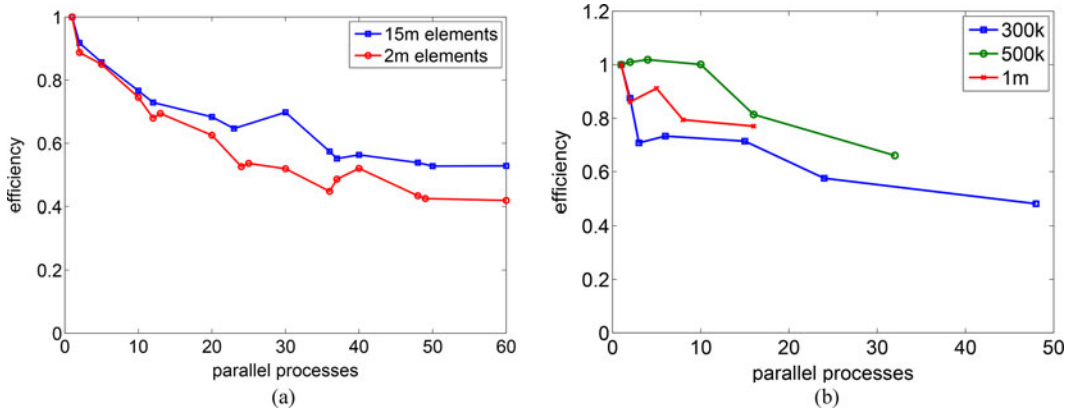

Fig. 3. Strong and weak scaling of the system matrix assembly—As expected, the efficiency of the matrix assembly drops when we move to more processors since the ratio of computation to communication is getting smaller. On the larger mesh, this effect is slightly smaller than on the coarse mesh [see Fig. 3(a)]. For Fig. 3(b), meshes were chosen such that the average load per process was approximately 300 000, 500 000 or one million elements. For half a million elements per process, the ratio of communication over computation appears to scale best.

TABLE III
TOTAL TIME REQUIRED TO ASSEMBLE THE SYSTEM MATRIX (ALL TIMES ARE IN SECONDS)

Mesh element no	2 m	15 m
1 process	27.89	231.41
2 processes	15.71	126.07
5 processes	6.56	54.04
10 processes	3.74	30.19
20 processes	2.23	16.93
40 processes	1.34	10.27
60 processes	1.11	7.3

G. Solver

Using the optimal multigrid preconditioning, we can use all Krylov subspace solvers available in PETSc, which are conjugate gradients (CG), generalized minimal residual (GMRES), and two CG algorithms for nonsymmetric problems BiCG and BiCGstab. Since we have a positive symmetric system matrix, we use CG. The stopping criterion for the CG solver was set to a relative residuum of 10^{-12}. On the two million element mesh, we observed large and reproducible fluctuations in the efficiency [see Fig. 5(a)], which most likely emerge from different communication requirements and cache usage for the different partitions. This effect is less visible on the large mesh since the computation load is proportionally larger than the access to the memory and communication between processors. The absolute run times on 60 processors were 1.3 s on two million elements and 9.6 s on 15 million (see Table VI). The weak efficiency [see Fig. 5(b)] indicates that the optimal load per processor is around 0.5 million elements. Comparing the strong scaling and the weak scaling, we observe that the weak scaling is worse. The reason for this is that the CG solver generally has a slower convergence rate on the larger meshes (see Table V). The required iterations on larger meshes correlate very well with the decrease in efficiency of the weak scaling [see Fig. 5(b)].

A possible alternative to using an AMG preconditioned CG algorithm is to set up a direct solver. A direct solver takes very long to assemble, but reduces subsequent solutions to mere forward and back substitutions. Thus, if many forward solutions are required (i.e., many electrodes are used) then a direct solver might be faster. PETSc interfaces to the MUMPS direct solver [25], using it as a preconditioner for the CG solver. This reduces each solution to one or two iterations. We found that the assembly scales very badly to larger meshes and the CG solver with MUMPS preconditioning does not scale well on many processes (see Table VII). This observed weak scaling of MUMPS is much worse than that of the MG preconditioners, meaning that for large problems, the number of required forward solutions for the direct solver to be faster increases (see Table VIII). Furthermore, the strong scaling is worse than that of ML as well, such that for many parallel processes AMG is always the better choice (as indicated with the minus symbol in the last row in Table VIII).

We can conclude that, for our application, a direct solver is only worth considering on relatively small problems with many electrodes, leading to more than hundred independent current injections. Most applications will be solved faster by using ML as a preconditioner.

H. Jacobian Calculation

The Jacobian matrix is computed based on the adjoint field method (7). In our implementation, the computations are handled by a struct `JacobianRowCalculator`, which computes the local stiffness matrices of all elements in the constructor and stores them. When the solver is then iterating over all lines of the measurement protocol, the member function `JacobianRowCalculator.getJacobianRow` is called with the voltage distributions for both drive and measurement current as arguments. `getJacobianRow` is then iterating over all elements and computing two matrix-vector products in each step to obtain the local entry of the row of the Jacobian matrix. This process requires no communication between processors at all, meaning we would expect a very good parallel efficiency. We observed that the efficiency is reliably larger than 1, going up to more than 2 in one case [see Fig. 6(a)]. The computation

TABLE IV
TIMES FOR THE ASSEMBLY OF THE PRECONDITIONER AND FOR THE SUBSEQUENT CG SOLUTION (ALL TIMES ARE IN SECONDS)

	2 m elements				15 m elements			
	Assembly		Solving		Assembly		Solving	
	Trilinos ML	BoomerAMG	Trilinos ML	BoomerAMG	Trilinos ML	BoomerAMG	Trilinos ML	BoomerAMG
5 processes	1.1	4.4	3.5	4.4	13.8	55.0	79.0	64.7
10 processes	0.7	3.3	2.3	3.1	5.8	34.1	45.8	43.5
20 processes	0.62	5.54	0.60	2.34	3.3	36.7	22.3	24.7
40 processes	1.02	8.3	0.58	2.1	3.7	41.5	13.0	14.4
60 processes	4.8	10.5	1.3	2.53	4.5	39.4	9.6	10.5

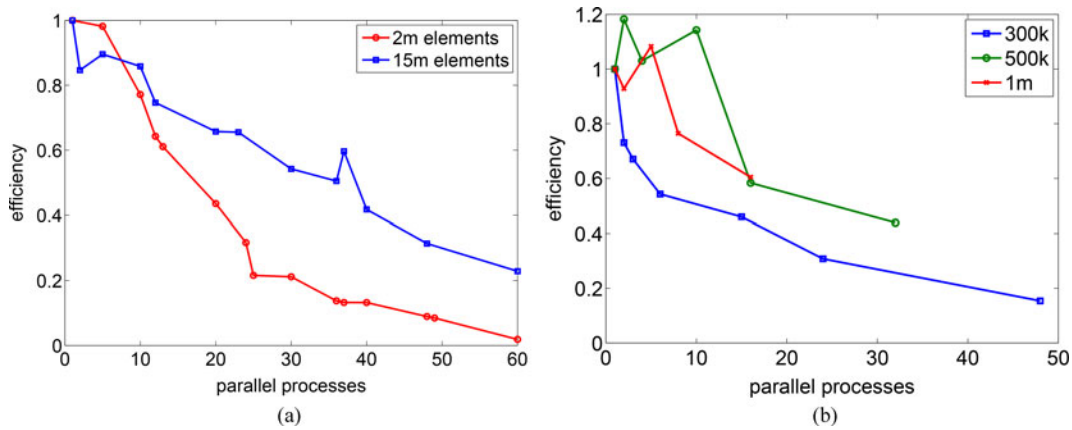

Fig. 4. Strong and weak scaling of the assembly of the AMG preconditioner ML—Since the computational cost is larger on the fine mesh, the increasing communication volume between parallel processes has a smaller influence on the overall efficiency of the preconditioner assembly when compared to the coarse mesh [see Fig. 4(a)]. For Fig. 4(b), meshes were chosen such that the average load per process was approximately 300 000, 500 000, or one-million elements. The weak scaling is a measure of how the ratio of communication over computation behaves when the code is applied to larger problems. With an average load of half a million elements, the weak efficiency scales best for the ML preconditioner assembly. We see, however, that the complexity of the larger meshes reduce the weak efficiency significantly.

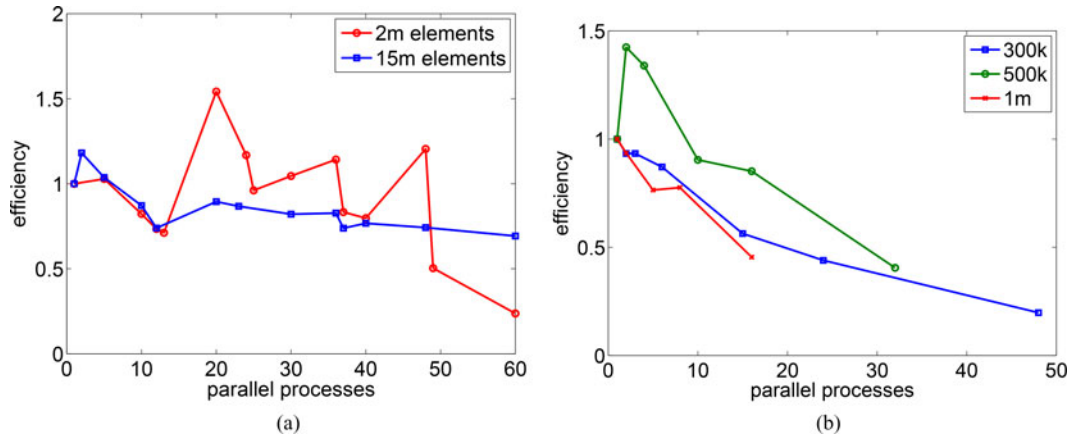

Fig. 5. Strong and weak scaling of the assembly of the CG solver with ML preconditioning—On the small mesh, the run times show reproducible large fluctuations, which are most likely caused by different cache efficiency for different partitions [see Fig. 5(a)]. On the larger mesh, where the computational load is larger these cache effects are not visible. For Fig. 5(b), meshes were chosen such that the average load per process was approximately 300 000, 500 000, or one million elements. An average load of half a million elements per processor leads to the optimal ratio of computation over communication. However, there is a significant drop in efficiency on many parallel processes. This can be explained by different convergence rates of the solver on the different meshes we used, as can be seen in Table V.

TABLE V
ITERATIONS OF THE CG SOLVER ON DIFFERENT MESH SIZES—THE NUMBER OF ITERATIONS ARE NOT DEPENDENT ON HOW MANY PARALLEL PROCESSES WERE USED

Element no	335 k	656 k	1 m	2 m	5 m	8 m	15 m
Iterations	44	41	34	35	46	43	63

TABLE VI
TIME TAKEN BY THE CG SOLVER WITH ML PRECONDITION TO COMPUTE ONE FORWARD SOLUTION (ALL TIMES ARE IN SECONDS)

Mesh element no	2 m	15 m
1 process	18.5	399
2 processes	8.3	169
5 processes	3.6	77
10 processes	2.25	45.8
20 processes	0.6	22.3
40 processes	0.58	13.0
60 processes	1.3	9.6

TABLE VII
PERFORMANCE OF THE MUMPS DIRECT SOLVER (ALL TIMES ARE IN SECONDS)

	2 m assem.	2 m solve	15 m assem.	15 m solve
1 proc.	1388.5	3.5	91375	48.9
10 proc.	258.0	0.94	28969	15.9
50 proc.	128.9	1.0	9469	16.3

TABLE VIII
NUMBER OF REQUIRED CONSECUTIVE FORWARD SOLUTIONS FOR MUMPS TO BE FASTER THAN ML—ON 50 PROCESSES ML IS FASTER IN BOTH, SETUP AND SUBSEQUENT SOLUTIONS

Mesh element no	2 m	15 m
1 process	93	261
10 processes	197	969
50 processes	–	–

time decreases up to the tested 60 parallel processes (see Table IX). This is most likely due to a more efficient use of cache memory. A reason to suspect this is that for the larger mesh the efficiency keeps improving for more processors while it remains around 1.4 for the smaller mesh, indicating that already all local stiffness matrices are stored in cache. The weak scaling of the Jacobian matrix computation [see Fig. 6(b)] indicates that the optimal load per processor is around half a million elements.

I. Verification of Correct Performance

The correctness of the results of the forward problem was verified in two ways. First, simulations were done on a mesh of a cube of varying sizes, number of elements, conductivities, and contact impedances of the two electrodes, which were placed on opposite sides of the cube. The results were then compared to the analytical solution and were precise up to computer precision. To make sure that it worked correctly also with more complicated shapes, the results of simulations on a head-shaped mesh were compared both to the version of EIDORS currently used in our group and to real measurements in a saline-filled tank. They matched the computed results by computer precision and the experimental results closely. Fig. 7 shows the resulting simulated electric potential distribution when a current is applied from the front of the head to the back of the head. It is visible how the potential drops at the highly resistive skull.

J. Application of the Solver to a Stroke Feasibility Simulation Study and to Imaging of Fast Neural Activity in a Live Rat

To illustrate the range of applications, we envisage for the presented software, we highlight two works using this solver. The first is a simulation study evaluating the feasibility of detecting two different types of stroke in the human head using EIT measurements at different frequencies for the injection current [7]. In the second application, the solver was used to compute the forward solutions and the Jacobian matrix on a 7-million element mesh of the rat brain, in order to reconstruct neural activity from EIT measurements on a living rat's brain [17].

The main difficulty in stroke type detection with EIT is that the finite-element model never accurately matches the measurement setup. These modeling errors can introduce large artifacts into the reconstructed images. To distinguish the main sources of artifacts, we simulated boundary voltages in the presence of three different modeling errors on a fine mesh and reconstructed the images on a coarse, modeling error free mesh. We had to compute forward solutions at 12 different frequencies for three different modeling errors with two different standard deviations each, and this for ischaemic and haemorrhagic stroke at two different locations in the head. This means that the study involved the computation of $288 \cdot N$ forward solutions, where N is the number of independent current injections (in our case 31). To compute that many forward solutions on a 5-million element mesh in MATLAB would have taken $288 \cdot 30$ min. This estimate shows that simulation studies of this scale were previously not feasible. Using the presented solver, the time for the forward simulations was reduced to $288 \cdot 1.7$ min on a workstation with two eight-core 2.4-GHz Intel Xeon CPUs with 20-MB cache each.

For EIT applications with high precision requirements, a very fine mesh is required for the forward computations. In the second application, the aim is to image fast neural activity in the rat cortex using a planar electrode array, which is surgically applied directly to the brain surface. A convergence study on the required finite-element size was performed as follows. Iteratively, the element size was reduced and ten meshes were created using the same settings. Then, the differences in simulated voltages on these meshes were compared to the differences of the next coarser meshes. Once the variability between meshes of the same resolution was of the same size than the variability between different mesh sizes, the optimal mesh size was reached (see Fig. 8). We found that the required mesh size for this application is 8-million elements. Due to the large number of electrodes, the measurement protocol (and thus, the Jacobian matrix) was very long and to compute all forward solutions and the

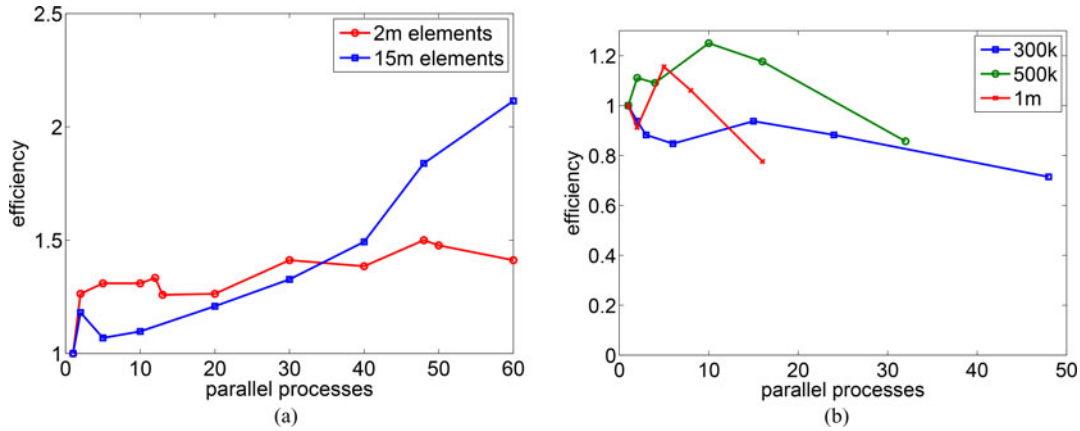

Fig. 6. Strong and weak scaling of the computation of one row of the Jacobian matrix—The efficiency of the computation of a Jacobian row increases, as we increase the number of processors [see Fig. 6(a)]. This is most likely due to a more efficient usage of the cache. On the larger mesh, where the computational load is larger these cache effects are more visible on many processors. For Fig. 6(b), meshes were chosen such that the average load per process was approximately 300 000, 500 000, or one million elements. The optimal load per processor is around half a million elements. We observe that the weak efficiency decreases on the larger meshes, which is most likely due to the memory access. When the processes are not distributed over all cluster nodes evenly the weak efficiency drops much earlier, which supports this claim.

Fig. 7. Simulated electric potential in a 2-million element mesh of the head —A current is applied from the front to the back of the head and the computed voltage distribution on a slice though the head mesh is shown. As expected, the electric potential drops sharply at the skull due to its low conductivity.

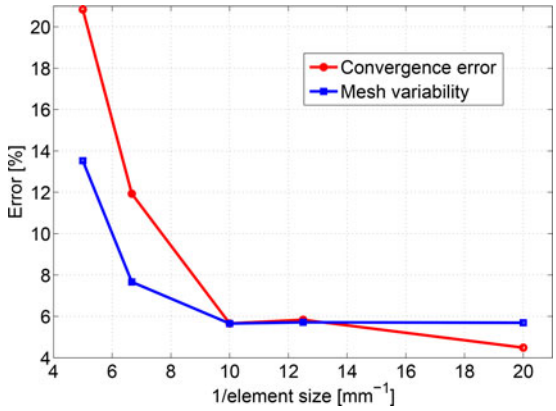

Fig. 8. Convergence of simulated voltages by reducing element size—The maximal relative error in simulated boundary voltages on meshes with the same element size (variability) was compared to the same error between meshes with different element size (convergence). 0.1 mm was found to be the optimal element size for this application.

Jacobian matrix in MATLAB would have taken around 16.5 h. By using the new parallel forward solver, this was reduced to just about an hour on the same workstation used for the first application. Therefore, we were suddenly able to make reasonably quick informed decisions about the quality of acquired data and experimental paradigm changes. The iterative process of improving experimental procedures was sped up by more than a factor of 16 and for the first time high-quality images of fast neural activity in the rat cortex using EIT could be reconstructed with the methods outlined in [17] (see Fig. 9).

IV. Performance

In this section, we evaluate the performance of the solver on a more general level by looking at the total run time for a typical EIT problem, comparing it to EIDORS and by comparing first-order elements to second-order elements.

A. Total Run Times With First-Order Elements

A common forward problem in EIT with pairwise current injection requires around 60 forward solutions for the unique drive and measurement current injections and around 1000 current protocol steps, i.e., 1000 lines in the Jacobian matrix. This means that we can estimate the total runtime for the solver by adding up the times for the single components, which have to be done once per execution (loading mesh partitions, finding electrode elements and areas, system matrix assembly, assembly of the preconditioner, computing local stiffness matrices used for the Jacobian row calculations) with 60 times the time it takes for one forward solution and 1000 times the time for one Jacobian row computation. Based on the run times shown in the previous sections, we estimated the total runtime for a common application of the solver (see Table X). We found that the overall efficiency scales very well (see Fig. 10).

B. Comparison to EIDORS

Since our group, like most others working in EIT, is currently using EIDORS for the computations of the forward model, we briefly want to compare the performance of the new parallel

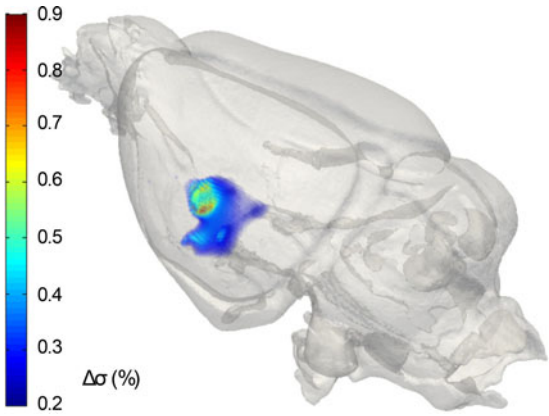

Fig. 9. Three-dimensional image of conductivity increase due to neural activity—This image shows the conductivity increase caused by opening ion channels during neural activity, which was induced by whisker stimulation of a rat. The activity patterns match the literature and correlate with intrinsic optics and local field potentials.

TABLE IX
TIME TAKEN FOR THE COMPUTATION OF ONE ROW OF THE JACOBIAN MATRIX (ALL TIMES ARE IN SECONDS)

Mesh element no	2 m	15 m
1 process	1.44	20.3
2 processes	0.57	8.6
5 processes	0.22	3.8
10 processes	0.11	1.85
20 processes	0.057	0.84
40 processes	0.026	0.34
60 processes	0.017	0.16

TABLE X
TOTAL ESTIMATED RUN TIMES FOR A PROTOCOL WITH 1000 LINES (ALL TIMES ARE IN SECONDS)

Mesh element no	2 m	15 m
1 process	3089.5	48644
2 processes	1335.6	22457
5 processes	498.2	6714.6
10 processes	254.3	3357.8
20 processes	117.1	1948.6
40 processes	78.3	1071.1
60 processes	86.7	842.4

solver with EIDORS on MATLAB. The new EIDORS version 3.7.1 has recently been released and was used by us to do the timings. For this section, we always compare the performance of EIDORS on a 2-million element mesh of the head with the performance of the DUNE solver on the same mesh in serial (see Table XI). To make the comparison valid, we disabled MATLAB's multithreading routines by calling `maxNumCompThreads(1)`. TheMATLAB version used was R2013a.

The standard solver of EIDORS is MATLAB's backslash operator, which takes approximately 1936 s for the direct solver to be assembled and around 12.5 s for each unique current pattern solved with it. The mumps direct solver is faster for each solve (3.5 s) as well as for its assembly (1389 s). Comparing iterative

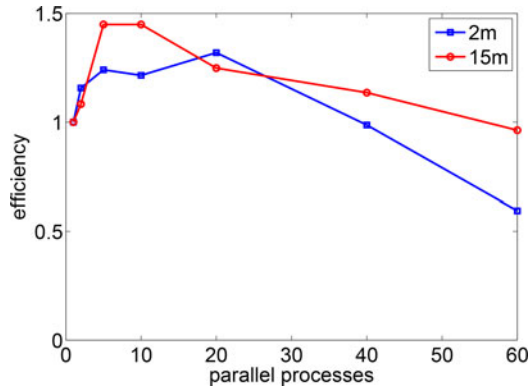

Fig. 10. Estimated efficiency of the total run time of a realistic EIT protocol—Shown in this figure is the efficiency based on the estimated run times shown in Table X. The solver scales very well on more processors, due to the very good scaling of the Jacobian matrix calculation, which accounts for most of the run time in serial.

TABLE XI
COMPARISON OF EIDORS/SUPERSOLVER IN MATLAB AND THE PRESENTED C++ SOLVER (PEITS) (ALL TIMES ARE IN SECONDS FOR AN EXECUTION ON ONE PROCESSOR)

	MATLAB	PEITS
Matrix assembly	128	27.8
Preconditioner assembly	0.8	5.4
CG solve step	39	18.5
Jacobian row calculation	0.3	1.4

solvers, we are using an incomplete LU decomposition as a preconditioner for a conjugate gradient solver in MATLAB and the ML AMG preconditioned CG solver in DUNE, since these two combinations turned out to be the best for the respective solver. While the MATLAB routine `ilu` was very quick (0.8 s), each successive solve with `pcg` took 39 s. In DUNE, the assembly of the AMG preconditioner took 5.4 s and each solve 18.5 s.

The assembly of the system matrix takes 128 s in EIDORS. However, this is difficult to compare to the assembly of the DUNE solver since EIDORS creates more data structures for later use (plotting, inverse, ...) and also assigns the electrode areas and ground indices differently. Our solver is less flexible and focuses only on the forward problem, which is one of the reason why it is more than twice as fast for the matrix assembly (27.8 s).

The mesh is loaded much faster in MATLAB (0.5 s) than in our solver (61 s), which is because MATLAB uses a compressed binary data format (.mat file), whereas our solver is currently still using ASCII data files. We are looking to switch to binary data files in future.

The calculation of a single row of the Jacobian is very difficult to compare since EIDORS and our solver use completely different approaches. While our solver uses the adjoint field method, EIDORS applies the derivative form [26]. Thus, we compare the total time it takes to compute a Jacobian matrix with either 1, 7 or 259 lines. In EIDORS, it took 2433, 2442, and 2817 s, respectively. Our solver took 1.44, 10.1, and 373 s. We see, that

the matrix-based approach of EIDORS is less dependent on the number of protocol lines. However, the memory usage is much higher and becomes inhibitive for bigger meshes and longer protocols. For 2-million elements and 258 protocol steps, the memory usage of EIDORS during the Jacobian calculation was 150 GB. This is why we also measured the time to compute the Jacobian with a MATLAB-based adjoint field method, which has been implemented in our group (the so-called SuperSolver used for instance in [9]). This implementation turned out to be extremely quick and much more memory efficient than the EIDORS implementation. The computation times for 1, 7, and 259 lines were 1.47, 4.3, and 77.3 s, which compares to our solver on approximately four parallel processes. The reason that it was significantly faster than our solver is that a generic derivative matrix is constructed. For successive protocol lines, it is then multiplied by the nodal potentials of the different forward solutions to get the gradients. Our solver on the other hand iterates over all elements and multiplies the local potentials with the gradients element wise for each protocol line.

To summarize, for a typical EIT application with 60 forward solutions and 1000 current protocol lines on a two-million element mesh with an iterative solver EIDORS takes around 6469 s and our DUNE solver on one processor around 3090 s.

C. Comparison With Elements of Second Order

It is very straight forward to switch to quadratic (or even cubic) shape functions. We only need to change one environment variable when compiling the solver. For smooth functions, the use of higher order shape functions achieves the same precision of the solution we get using first-order elements, but with a much smaller mesh. In order to get a rough estimate of the ratio of the required element sizes to get the same precision, we created cube-shaped meshes with regular tetrahedral elements with size $h = 1/10, 1/20, 1/30, \ldots$ for the cube with dimensions $(1, 1, 1)$. Then, we assigned two electrodes to the central square with area 1/25 on opposite sides of the cube and applied a current of 133 μA. The cube was assigned a uniform conductivity of 0.3 Sm^{-1}. We then plotted the convergence of the simulated voltage between these two electrodes with respect to the element size for first- and second-order polynomials and found that for a similar accuracy the number of elements can be reduced by a factor of 67 when second-order elements are used (see Fig. 11). The slope of both curves is the same, because the solution is in $H^1(\Omega)$ and the convergence rate of second-order elements is, therefore, the same than that of linear elements.

We do not know the problem dependence of this observed shift between the two curves in Fig. 11, which makes it impossible to generalize our finding. For some applications, it might be very useful to switch to second-order shape functions, for others, it might even slow down the code.

In our cube-shaped test example, we compared the run times of the different parts of the solver on the 24 576 000 linear element mesh with the run times on the 384 000 quadratic element mesh. The result on the small quadratic mesh was $v_{2\mathrm{nd}} = 0.267988771$ V and on the large mesh with linear shape functions $v_{1\mathrm{st}} = 0.267987305$ V, meaning the mesh with

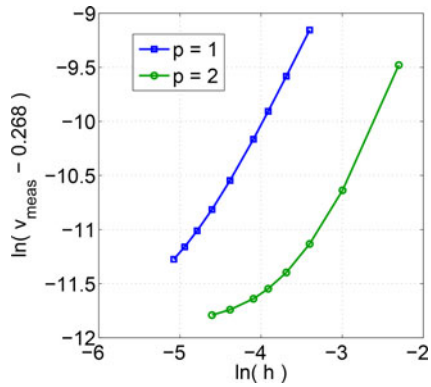

Fig. 11. Convergence of the simulated voltage with respect to the element size—By using quadratic shape functions the same convergence can be achieved with around 67 times less elements.

TABLE XII
RUN TIME COMPARISON FIRST- VERSUS SECOND-ORDER ELEMENTS (ALL TIMES ARE GIVEN IN SECONDS)

	Serial		20 processes	
	First	Second	First	Second
Loading partitions	357.7	4.78	21.33	0.42
Matrix assembly	372.4	39.8	112.02	6.76
AMG assembly	38.8	6.52	14.53	1.70
Solve	64.2	14.6	10.61	3.27
Jacobian row	9.94	0.32	0.59	0.02

second-order shape functions was a bit closer to the value the simulations converged to. All parts of the solver were much faster on the small mesh of second-order elements (see Table XII). This means that the use of second-order shape functions reduces the computation time to achieve a certain precision significantly in this test example. The PETSc preallocation was set to 2500 diagonal and 2500 off-diagonal entries per row to account for the electrodes on the 25-million element mesh.

V. DISCUSSION

We see from the results shown in the previous sections that our solver significantly reduces the time needed for the computation of forward solutions in EIT. To facilitate the use of the solver we provide MATLAB functions that write a mesh in DGF format and call the solver with different settings. This makes it possible to run the solver from a MATLAB code by calling just one MATLAB function `run_forward_solver()`, which returns the Jacobian matrix and the measured voltages.

The solver is actively used in our group for forward simulations on 5–15 million element meshes. The computed results are used for the image reconstruction from experimental data. Furthermore, the code is applied to forward models with changing settings such as electrode position, tissue conductivity, contact impedances and more, with the aim to identify different sources of image artifacts caused by modeling errors.

Especially in the use of adaptive mesh optimization, there is room for improvements, such as using second-order elements

in regions where the solution is in H^2 or refining the mesh around the electrodes based on local error estimates. DUNE-FEM natively supports such local grid refinements.

In comparison to MATLAB, our solver is already faster in serial in every step except for the computation of the Jacobian matrix. This is where we see the largest potential speed improvements, especially since this is the step that is repeated around 1000 times for a typical EIT application. We can think of several approaches that can speed up the Jacobian computation. One would be to use a coarser mesh where the Jacobian is computed on (as shown in [10] and [27]), since most reconstruction algorithms do not (and do not need to) rely on a fine mesh. Another one would be to run the jacobian matrix calculation on a GPU, where memory access is much faster. This has already been shown to improve the speed significantly by Borsic *et al.* [13].

REFERENCES

[1] B. Brown, D. Barber, and A. Seagar, "Applied potential tomography: Possible clinical applications," *Clinical Phys. Physiol. Meas.*, vol. 6, no. 2, pp. 109–121, May 1985.

[2] I. Frerichs, "Electrical impedance tomography (EIT) in applications related to lung and ventilation: A review of experimental and clinical activities," *Physiol. Meas.*, vol. 21, no. 2, pp. R1–R21, 2000.

[3] B. Brown, "Electrical impedance tomography (EIT): A review," *J. Med. Eng. Technol.*, vol. 27, no. 3, pp. 97–108, 2003.

[4] D. Holder and T. Tidswell, "Electrical impedance tomography of brain function," in *Electrical Impedance Tomography: Methods, History and Applications*, D. S. Holder, Ed. New York, NY, USA: Taylor & Francis, 2004, ch. 4, pp. 127–166.

[5] D. Holder, "Detection of cerebral ischaemia in the anaesthetised rat by impedance measurement with scalp electrodes: Implications for non-invasive imaging of stroke by electrical impedance tomography," *Clinical Phys. Physiol. Meas.*, vol. 13, no. 1, pp. 63–75, 1992.

[6] A. Tidswell, A. Gibson, R. Bayford, and D. Holder, "Validation of a 3D reconstruction algorithm for EIT of human brain function in a realistic head-shaped tank," *Physiol. Meas.*, vol. 22, no. 1, pp. 177–185, Feb. 2001.

[7] E. Malone, M. Jehl, S. Arridge, T. Betcke, and D. Holder, "Stroke type differentiation using spectrally constrained multifrequency EIT: Evaluation of feasibility in a realistic head model," *Physiological Meas.*, vol. 35, no. 6, pp. 1051–1066, Jun. 2014.

[8] A. Adler and W. Lionheart, "Uses and abuses of EIDORS: An extensible software base for EIT," *Physiol. Meas.*, vol. 27, no. 5, pp. S25–S42, May 2006.

[9] L. Horesh, M. Schweiger, M. Bollhöfer, A. Douiri, D. Holder, and S. Arridge, "Multilevel preconditioning for 3D large-scale soft field medical applications modelling," *Int. J. Inf. Syst. Sci.*, vol. 2, pp. 532–556, 2006.

[10] A. Borsic, R. Halter, Y. Wan, A. Hartov, and K. Paulsen, "Electrical impedance tomography reconstruction for three-dimensional imaging of the prostate," *Physiol. Meas.*, vol. 31, no. 8, pp. S1–S16, Aug. 2010.

[11] O. Schenk. (2014). PARDISO Website. [Online]. Available: http://www.pardiso-project.org/

[12] M. Soleimani, C. Powell, and N. Polydorides, "Improving the forward solver for the complete electrode model in EIT using algebraic multigrid," *IEEE Trans. Med. Imag.*, vol. 24, no. 5, pp. 577–583, May 2005.

[13] A. Borsic, E. Attardo, and R. Halter, "Multi-GPU Jacobian accelerated computing for soft-field tomography," *Physiol. Meas.*, vol. 33, no. 10, pp. 1703–1715, Oct. 2012.

[14] N. Gençer and I. Tanzer, "Forward problem solution of electromagnetic source imaging using a new BEM formulation with high-order elements," *Phys. Med. Biol.*, vol. 44, no. 9, pp. 2275–2287, Sep. 1999.

[15] E. Somersalo, M. Cheney, and D. Isaacson, "Existence and uniqueness for electrode models for electric current computed tomography," *SIAM J. Appl. Math.*, vol. 52, no. 4, pp. 1023–1040, 1992.

[16] N. Polydorides and W. Lionheart, "A MATLAB toolkit for three-dimensional electrical impedance tomography: A contribution to the electrical impedance and diffuse optical reconstruction software project," *Meas. Sci. Technol.*, vol. 13, no. 12, pp. 1871–1883, 2002.

[17] K. Y. Aristovich, G. S. dos Santos, B. C. Packham, and D. S. Holder, "A method for reconstructing tomographic images of evoked neural activity with electrical impedance tomography using intracranial planar arrays," *Physiol. Meas.*, vol. 35, no. 6, pp. 1095–1109, Jun. 2014.

[18] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger, "A generic interface for parallel and adaptive scientific computing: Abstraction principles and the Dune-Fem module," *Computing*, vol. 90, no. 3–4, pp. 165–196, 2010.

[19] T. Davis, "A column pre-ordering strategy for the unsymmetric-pattern multifrontal method," *ACM Trans. Math. Softw.*, vol. 30, no. 2, pp. 165–195, 2004.

[20] S. Balay, J. Brown, K. Buschelman, W. Gropp, D. Kaushik, M. Knepley, L. Curfman McInnes, B. Smith, and H. Zhang. (2014). PETSc Web page. [Online]. Available: http://www.mcs.anl.gov/petsc

[21] (2014). CGAL, Computational Geometry Algorithms Library. [Online]. Available: http://www.cgal.org

[22] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek, "Parallel hypergraph partitioning for scientific computing," presented at the IEEE 20th Int. Parallel & Distributed Processing Symp., Rhodes Island, Greece, 2006, p. 10.

[23] R. Falgout, A. Cleary, J. Jones, E. Chow, V. Henson, C. Baldwin, P. Brown, P. Vassilevski, and U. Meier Yang. (2014). Hypre website. [Online]. Available: http://acts.nersc.gov/hypre/

[24] M. Gee, C. Siefert, J. Hu, R. Tuminaro, and M. Sala, "ML 5.0 smoothed aggregation user's guide," Sandia National Laboratories, Albuquerque, NM, USA, Tech. Rep. SAND2006-2649, 2006.

[25] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, "Hybrid scheduling for the parallel solution of linear systems," *Parallel Comput.*, vol. 32, no. 2, pp. 136–156, 2006.

[26] T. Yorkey, "Electrical impedance tomography with piecewise polynomial conductivities," *J. Comput. Phys.*, vol. 91, no. 2, pp. 344–360, 1990.

[27] A. Adler, A. Borsic, N. Polydorides, and W. Lionheart, "Simple FEMs aren't as good as we thought: experiences developing EIDORS v3. 3," presented at the Conf. Electrical Impedance Tomography, Hannover, NH, USA, Jun. 2008.

Authors ' photographs and biographies missing at the time of publication.